

**AFRL-IF-RS-TR-2002-144**  
**Final Technical Report**  
**June 2002**



# **SYNTHETIC FORCE STRUCTURE SIMULATION (SFSS)**

**CACI Technologies, Incorporated**

**Sponsored by**  
**Defense Advanced Research Projects Agency**  
**DARPA Order No. 558T**

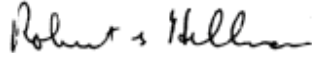
*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*


**The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.**

**AIR FORCE RESEARCH LABORATORY**  
**INFORMATION DIRECTORATE**  
**ROME RESEARCH SITE**  
**ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2002-144 has been reviewed and is approved for publication.

APPROVED:   
ROBERT HILLMAN  
Project Engineer

FOR THE DIRECTOR:   
MICHAEL L. TALBERT, Technical Advisor  
Information Technology Division  
Information Directorate

<b>REPORT DOCUMENTATION PAGE</b>			<i>Form Approved</i> <b>OMB No. 074-0188</b>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
<b>1. AGENCY USE ONLY (Leave blank)</b>		<b>2. REPORT DATE</b> JUNE 2002	<b>3. REPORT TYPE AND DATES COVERED</b> Final Feb 01 – Oct 01	
<b>4. TITLE AND SUBTITLE</b> SYNTHETIC FORCE STRUCTURE SIMULATION (SFSS)			<b>5. FUNDING NUMBERS</b> C - F30602-00-D-0221 Task 001 PE - 63789F/62702F PR - 407T TA - 1F WU - 02	
<b>6. AUTHOR(S)</b> Gary Blank				
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b> CACI Technologies, Incorporated 14151 Park Meadow Drive Chantilly Virginia 20151			<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b>  N/A	
<b>9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b> Defense Advanced Research Projects Agency AFRL/IFTC 3701 North Fairfax Drive Arlington Virginia 22203-1714			<b>10. SPONSORING / MONITORING AGENCY REPORT NUMBER</b>  AFRL-IF-RS-TR-2002-144	
<b>11. SUPPLEMENTARY NOTES</b> AFRL Project Engineer: Robert Hillman/IFTC/(315) 330-4961/ Robert.Hillman@rl.af.mil				
<b>12a. DISTRIBUTION / AVAILABILITY STATEMENT</b> APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				<b>12b. DISTRIBUTION CODE</b>
<b>13. ABSTRACT (Maximum 200 Words)</b> C + + library classes for Force Structure Simulation (FSS) based on the parallel simulation framework called SPEEDES (Synchronous Parallel Environment for Emulation and Discrete-Event Simulation) was developed. These classes derive from and/or utilize the SPEEDES Application Programming Interface (API), and the SPEEDES simulation methodology for execution on high-performance multiprocessor systems. These FSS classes were designed to be the foundation of a R&D simulation framework for researching predictive course of action analysis. The software will prove an easily extensional means to accommodate an increasingly detailed, powerful and elaborate assessment tool.				
<b>14. SUBJECT TERMS</b> Parallel Event Simulation, War Simulation, SPEEDES, STOW				<b>15. NUMBER OF PAGES</b> 15
				<b>16. PRICE CODE</b>
<b>17. SECURITY CLASSIFICATION OF REPORT</b>  UNCLASSIFIED	<b>18. SECURITY CLASSIFICATION OF THIS PAGE</b>  UNCLASSIFIED	<b>19. SECURITY CLASSIFICATION OF ABSTRACT</b>  UNCLASSIFIED	<b>20. LIMITATION OF ABSTRACT</b>  UL	

## TABLE OF CONTENTS

<b>1</b>	<b>INTRODUCTION.....</b>	<b>1</b>
<b>2</b>	<b>WARFARE MODELING FRAMEWORK .....</b>	<b>1</b>
2.1	Essential Elements of Warfare Modeling Framework.....	2
2.2	Define the scenario .....	3
2.3	Simulate Until End Time .....	3
<b>3</b>	<b>THE SFSS BASE CLASSES.....</b>	<b>3</b>
3.1	Public Objects .....	4
3.2	Assets .....	4
3.3	Fixed Assets.....	5
3.4	Mobile Assets.....	5
3.5	Commanders .....	5
3.6	Asset Subsystems.....	6
<b>4</b>	<b>TECHNICAL REFERENCE OF CLASS INTERFACES .....</b>	<b>7</b>
<b>5</b>	<b>A DEMO SIMULATION .....</b>	<b>7</b>
<b>6</b>	<b>FUTURE WORK/DIRECTIONS FOR SFSS .....</b>	<b>8</b>
6.1	Communications Modeling.....	8
6.2	Tactics .....	9
6.3	Tactical Picture .....	10
6.4	Plans.....	10
6.5	Logistics.....	11
6.6	MOE Classes.....	11

## 1 INTRODUCTION

SFSS is a set of C++ base classes and a host of accompanying infrastructure from which parallel warfare simulations can be built. One could, for example, use these generic classes to define various types of planes, ships, ground forces, commanders, fixed facilities, etc. and then simulate scenarios comprised of these entities.

Although moderate in scope, these classes were designed to be the foundation of a thoroughgoing framework for constructing warfare simulations. Thus, in addition to being functional in its own right, the software should prove to be easily extended to accommodate an increasingly detailed, powerful and elaborate library of classes.

The library of SFSS classes is based on the parallel simulation framework called SPEEDES (Synchronous Parallel Environment for Emulation and Discrete-Event Simulation). This means that some classes derive from and/or utilize SPEEDES classes, and that the SPEEDES methodology is used to coordinate concurrent activities taking place on multiple CPUs. These processors can be located in a high-performance multiprocessor, a group of connected workstations, or a combination of the two. Other advantages deriving from SPEEDES include:

- Optimistic processing and an accompanying library of rollbackable classes.
- Automated data distribution and filtering.
- The ability for separate programs to connect into the simulation, thus providing a “window” into the current state, and also allowing some external control.
- An HLA “gateway” that allows a simulation to participate in an HLA federation.
- Facilities for dynamically creating simulation objects during the course of the simulation.

## 2 WARFARE MODELING FRAMEWORK

As was mentioned above, the SFSS classes have been designed as part of a general scheme for doing warfare modeling. In this section, we present a brief overview of this scheme. Having the “big picture” in mind will serve two purposes: first, it will make clear how the existing SFSS classes form the basis of a force structure simulation framework, and second, it will point the way toward possible future extensions of this work.

## **2.1 Essential Elements of Warfare Modeling Framework**

In this section, we introduce the salient entities from which warfare simulations are constructed. These include commanders, assets, asset subsystems, measures of effectiveness, and the interaction database.

### **2.1.1 Commanders**

A commander is an entity that controls a set of assets and subordinate commanders. These forces are deployed by the commander in order to achieve one or more objectives. In addition to its controlled forces, commander objects have a plan, a tactical picture, and a tactics table. A plan consists of a sequence of actions the commander intends to carry out, the purpose of which is to accomplish one or more of the commander's objectives. The tactical picture represents the commander's *perception* of the current situation (as opposed to the actual situation). It integrates *a priori* knowledge and information from sensor reports into a unified picture, some of which may be incorrect. The tactics table contains a set of rules that tell the commander how to react to situations represented in the tactical picture. Applying such rules to the current tactical picture enables the commander to dynamically respond to events as they unfold during the course of the simulation.

### **2.1.2 Assets**

An asset is the basic fighting unit in a simulation. The two main types of assets are fixed assets and mobile assets (i.e. stationary and movable units). Examples of the former include airfields, sea ports, radar facilities, and fixed SAM sites. Examples of the latter include planes, ships, and tanks.

### **2.1.3 Asset Subsystems**

An asset subsystem is an entity that provides a capability to an asset. Examples include sensors, weapons, and communications devices.

### **2.1.4 Measures of Effectiveness (MOEs)**

An MOE is a metric collected during the simulation summarizing how well (or poorly) objectives have been achieved. Examples include the number of enemy assets killed, the number of friendly assets killed, the ratio of enemy assets killed to friendly assets killed, the number of engagements, the probability that some objective is achieved, the time required to achieve an objective, etc.

### **2.1.5 Interaction Database**

The Interaction Database contains information needed to simulate the interaction between certain types of objects. A sensor probability of detection (Pd) table, for example, might be included in the database. This table would specify the probability that a given type of sensor detects an object, based on the object type and the distance from the sensor (and possibly other factors).

## **2.2 Define the scenario**

Assume that there are two opposing alliances, referred to as “Red” and “Blue.” Defining the scenario means specifying all the Red and Blue assets, building a hierarchy of commanders, assigning assets to the commanders, and devising plans for some or all of the commanders. This constitutes the initial state of the simulation. The initialization data can be read in from input files or entered using a graphical interface.

## **2.3 Simulate Until End Time**

Once the initial simulation state has been entered, execution can begin. Commanders deploy forces according to their plans. Sensors detect, identify, and classify objects, and report to commanders. This information is integrated into the commanders' tactical pictures.

Simulation events are generated by commanders executing plans. Also, as the tactical picture evolves, tactical rules are activated, thus generating other events. These events alter the simulation state, changing the tactical pictures of commanders, thus generating more events. This cycle repeats until the simulation end time is reached.

Meanwhile, MOE data is collected as the simulation is running. When the end-time of the simulation arrives, the MOEs are calculated and output. This is the main purpose of the whole exercise--to measure various effects and outcomes of a simulated scenario in order to get a feel for what would happen if the scenario were to take place in the real world.

## **3 THE SFSS BASE CLASSES**

We have developed a set of generic C++ classes from which warfare simulations like those described above can be constructed. These are SPEEDES simulation objects, so instances of derived classes can be distributed among two or more processors (and their activities logically coordinated), thus producing a parallel simulation. This relieves the programmer from the difficult task of managing the communication and coordination of processes running concurrently on many CPUs.

The following sections provide descriptions of the most significant of the SFSS classes. To help understand these descriptions, a few key terms must first be defined:

- **Publish:** When a simulation object publishes itself, it announces that it is making part of its state publicly viewable by other simulation objects that are interested in this information.
- **Subscribe:** This is the complement of publish, except done with respect to object classes. That is, when a simulation object subscribes to a class, it is asking SPEEDES to inform it of the existence of all simulation objects of that type, and to provide updates of the state of these simulation objects.

- **Attribute:** The elements of a simulation object's state that are to be made public for subscribers are called its attributes. For example, a ship simulation object could have a name, id number, position, course, and speed attributes (among others). Note that a simulation object's attributes are just the part of the object that it wishes to advertise, and not necessarily the entire state of the simulation object.
- **Discover:** When a simulation object is created, all subscribers to that class are informed of the existence of the new simulation object. This is called discovering a simulation object.
- **Reflect:** When a simulation object modifies one or more of its attributes, subscribers to that class are informed of the new attribute values. The subscribers are said to reflect the updated state of the object.

### 3.1 Public Objects

This is the base class for classes that need to publish themselves and/or subscribe to other classes. This class (called `S_PublicObj`) automatically publishes itself, and contains four (public) attributes for identifying the object: a name, a type, an object ID, and an alliance. The name is a character string; the other three attributes are integers. These attributes are assigned by the user to identify public objects. It is assumed that the name uniquely identifies the object (no two objects should have the same name), as does the (type object ID) pair. The alliance attribute specifies which side the object is allied with (e.g. Red, Blue, or White).

Public objects can subscribe as well as publish. The `S_PublicObj` class has utilities for looking up information about discovered objects.

### 3.2 Assets

This class (called `S_Asset`) is the base class for building classes to model military assets. Since it inherits from `S_PublicObj`, it publishes itself. In addition to the identifying attributes inherited from `S_PublicObj`, the asset class adds the following attributes: a commander, a status, a sensor list, and a weapon list. The commander refers to the asset-level commander controlling this asset (e.g. the pilot in a plane). The status is an integer attribute that can be used as desired to indicate the state of the object (e.g. 1 for operational, 0 for non-operational, etc.). The sensor list attribute is a list of sensor subsystems owned by the asset. The weapon list attribute is a list of weapon subsystems on the asset. The list attributes can be empty (i.e. contain 0 items) or arbitrarily large.

Since assets represent fighting units, they must be able to engage other entities and model what happens when they are struck by a weapon. The `S_Asset` class contains a virtual method for engaging other assets and one for simulating weapon hits. Derived classes must implement these virtual methods in order to customize the behavior of the asset.

An event has been defined that calls the “engage” method. This means that other objects can schedule an engage event on an asset. A commander object can use this mechanism



to order an asset to engage a target. Likewise, an event has been defined that calls the “weapon hit” method. This event can be scheduled on an asset involved in an engagement. For example, if an asset fires a weapon at a target and determines that the weapon actually struck the target, it schedules a “weapon hit” event on the target. The target would assess the damage, and possibly return fire.

### 3.3 Fixed Assets

This class (called `S_FixedAsset`) inherits from `S_Asset`, and augments it with a location attribute. The location contains the latitude, longitude, and altitude of the object. This class generally will be used to build classes representing objects that have an absolutely fixed location, such as an airfield or a seaport. However, since the location attribute can be changed, this class can be also used to represent objects that move infrequently. In these cases, changing the location will result in a discontinuous jump from one position to another. Although this is totally unrealistic, in some circumstances it may be acceptable to do this. In all other cases--where a continuous motion model is needed--the mobile asset class is the one to use.

### 3.4 Mobile Assets

This class (called `S_MobileAsset`) inherits from `S_Asset`, and augments it with a motion script attribute. This attribute defines a continuous *track* that specifies the exact position of the asset over an interval of time. The motion track is defined by a sequence of great-circle *legs* from point  $P_0$  to  $P_1$ ,  $P_1$  to  $P_2$ ,  $P_2$  to  $P_3$ , etc., where each point is a (latitude, longitude, altitude) triple.

The mobile asset class has methods for setting or altering the motion script, and for calculating the location of the asset at a given time. Also, events have been defined that alter the motion script of the mobile asset on which it has been scheduled. These can be used by commanders to vector a mobile asset to a location or sequence of locations (that is, down an entire track).

### 3.5 Commanders

The `S_Commander` class inherits from `S_PublicObj` to which it adds three data members: a plan, a tactical picture, and a tactics member (note that these are not public attributes). These three members correspond to the capabilities described above. Every commander should be able to devise a plan, keep a representation of his knowledge of the current situation, and use tactics to respond dynamically to the current situation.

Of these three capabilities, only the planning has been developed to a significant degree in the current SFSS version. The plan member keeps a time-stamped, ordered list of actions that the commander intends to carry out. The `S_Commander` class has methods for adding actions to the plan, and for executing plan actions. Also, there are methods for planning two specific actions: vectoring a mobile asset and ordering an asset to engage a target.

Two subclasses of `S_Commander` have been implemented: `S_MissionCommander` and `S_AssetCommander`. The former represents a high-level commander directing an entire mission area such as air warfare, surface warfare, strike warfare, etc. The latter represents the commander of an individual asset, such as a pilot or ship captain.

The mission commander class adds three (public) attributes to the `S_Commander` class: an integer indicating his mission area, a list of controlled assets, and a list of subordinate commanders. This reflects the fact that a mission commander is in charge of a group of assets and subordinate commanders. The `S_MissionCommander` will make plans to deploy its assets and subordinate commanders, and will have high-level tactics that direct these forces.

The `S_AssetCommander` class is just an `S_Commander` with an asset reference (public) attribute. This attribute provides a “handle” for identifying the asset it commands and for scheduling events on it. While developing the code, serious questions arose regarding the wisdom of having separate simulation objects represent the asset and the commander. In a future version, we may wish to fuse the two into a single object.

### 3.6 Asset Subsystems

Simulation developers can create *private* subsystems without having to do anything special (above and beyond the usual SPEEDES methodology regarding rollbackability, etc.). However, if there are attributes within the subsystem that need to be public, a specific procedure must be followed to create such classes.

An asset subsystem is basically just a sub-object that contains public attributes; this sub-object is part of, or controlled by, an asset object. Examples of asset subsystems include sensors, weapons, and communications systems. Each of these has its own attributes and is part of an asset such as a plane or a ship. There are two ways that a subsystem is incorporated within an asset: it can be an attribute of the asset or else an item in a list attribute that is part of the asset. For example, the `S_Asset` class has a list of sensors and a list of weapons. Each of the sensors in the list is itself a subsystem (or sub-object) that has public attributes.

The `Subobject` class has been created in order to help build public asset subsystems. The `Subobject` class has two attributes--an integer type and integer ID--that are published automatically. These are provided to identify the subsystem.

To create subsystem classes, the developer needs to have the class inherit from the `Subobject` class and follow the procedure for creating classes inheriting from the `SPEEDES OBJECT_ATTRIBUTE` class. Examples of how to do this are provided by the `Sensor` and `Weapon` classes.

The `Sensor` class is a generic base class for building sensors. It inherits the type and ID attributes from the `Subobject` class and adds a range and status attribute. The range is

the maximum distance that the sensor can detect objects. The status is an integer attribute that can be used as desired (e.g. to denote whether the sensor is operating or not).

The Weapon class is a generic base class for building weapons. It inherits the type and ID attributes from the Subobject class and adds three more: a range attribute, a weapon speed attribute, and a status attribute. The range is the maximum distance from the target that the weapon can be used effectively. The weapon speed denotes the average speed of the weapon moving to the target. The status is an integer attribute that can be used as desired (e.g. to denote whether the weapon is operating or not).

## **4 TECHNICAL REFERENCE OF CLASS INTERFACES**

The “Synthetic Force Structure Simulation Class Reference Manual” documents the public and protected interfaces of the SFSS classes. For each class, there are two sections, one giving a brief description and another giving a detailed description. The classes are arranged alphabetically, but there is also a hierarchical index. The latter is most reasonable way to approach the mass of documentation.

The classes can be divided into a few groups:

1. Simulation objects: These are the most important classes; they are classes for building public simulation objects. All descend from the SPEEDES S\_SPHLA class, and have names that begin with S\_ (which denotes the fact that they represent simulation objects).
2. Proxies: For each simulation object class, there is a corresponding proxy class to represent simulation objects of this type. This is what subscribers receive when they discover instances of these classes. Also, there is a proxy class representing SIMOBJ\_REFERENCE attributes. These attributes contain a series of IDs needed for referring to simulation objects.
3. Object Attributes: These are attributes which themselves contain public attributes. All inherit from the SPEEDES OBJECT\_ATTRIBUTE class. Included in this group are Subobject classes (sensors and weapons), and SIMOBJ\_REFERENCE classes.
4. Supporting Classes: This group includes the plan class, action classes, track point classes, and the announcement message class (used by assets to declare themselves to commanders).

## **5 A DEMO SIMULATION**

In order to demonstrate how to use the SFSS base classes, a small simulation was assembled. Building on SFSS, classes representing airfields, jets, missiles, and an air warfare commander were created. Forces are divided into two alliances, a Red side and a Blue side.

The simulation scenario runs as follows:

1. During the initialization phase, input data is read, and simulation objects are created and initialized. The scenario has two Red airfields, five Blue jets (each carrying missiles), and one Blue air warfare commander.
2. The Blue air warfare commander creates a plan to strike the two Red airfields. This consists of sending one jet to attack one of the airfields, and another to attack the other airfield.
3. The Blue air warfare commander executes his plan. This vectors the two jets toward the two Red airfields, and later issues engage orders to both.
4. The jets take off and fly along their assigned tracks. When given the order, they each engage their respective targets. This causes each to fire two missiles at the airfield. A random draw determines if a missile succeeds in hitting the target. Each time a missile hits, an event is scheduled on the target. This event determines how much of the airfield has been damaged, and how much is still operational. The percentage of the airfield's capability that is operational is assigned to the object's Status attribute. Subscribers to airfields can therefore “see” the damage level of each airfield.

Please refer to the “SFSS Demo Reference Manual” for a detailed description of the interfaces for the airfield, jet, missile, and air warfare commander classes. This report is in the same format as the one describing the SFSS classes in the Synthetic Force Structure Simulation Class Reference Manual

## **6 FUTURE WORK/DIRECTIONS FOR SFSS**

This report began with an overview of the SFSS classes, describing in general terms what they are and how they can be used to build parallel warfare simulations. Then, a simple example simulation was detailed to show some specifics of how the SFSS classes can be used. Finally, a detailed description of the class interfaces was presented.

The point of these prior sections was to show how SFSS provides the basics for building parallel simulations. In this section, we wish to supply some ideas for ways in which SFSS can be augmented with additional capabilities in order to make it more powerful and easier to use.

### **6.1 Communications Modeling**

Currently, the SFSS classes do not contain any communications modeling. For example, a commander can vector a mobile asset simply by calling a method (which, in turn, schedules an event on the asset object). In reality, there must be some kind of communications link in order for a commander to issue an order to an asset. There are bandwidth limits, delays, and reliability issues associated with any real-world

communication system. These factors can have significant consequences in many scenarios. Therefore, one possible extension for SFSS would be to include some form(s) of communications modeling. Ideally, there would be two or more levels of resolution in the model classes. This way, developers could choose the degree of communications complexity (or “resolution”) required by their simulation. At one end of the spectrum, there could be very detailed communications modeling; at the opposite end, there would be little or none.

Since AFRL is developing communications models, it may be possible to leverage this work by integrating all or part of this work into SFSS.

## 6.2 Tactics

In order to be able to respond to dynamically evolving situations, commanders need a set of rules to guide them. The `S_Commander` class has a generic `TacticsObject` member for this purpose, but the class is stubbed out. What is needed is an object in which developers can insert tactics rules that will be activated when relevant situations arise.

The general form of a tactical rule is:

*IF Condition THEN Action;*

where *Condition* is a Boolean expression, and *Action* is a function or method. The meaning of this is: whenever *Condition* is satisfied by the current situation, execute *Action*. Building a coherent ensemble of these simple IF-THEN rules is actually a very general mechanism (as powerful, in fact, as the usual programming languages), and forms the basis of so-called “expert systems,” which use large collections of such rules to mimic the subtle behavior of an expert. The point is that by assembling a set of such rules, one can create arbitrarily sophisticated tactical responses. At the simplest level, each rule would have an associated “trigger-event” that would execute an action whenever the event was scheduled on the object (i.e. the occurrence of the event is, in effect, the *Condition* part of the rule).

At a higher level of complexity, each rule could have an integer precedence value associated with it. This way, if two or more rules are applicable to a given situation (i.e. their part *Condition* is satisfied), the one with the highest precedence is executed. A special case of this is a list of rules ordered from high to low precedence. The first rule whose *Condition* is satisfied would be selected for execution. At an even higher level of complexity, we encounter the idea of meta-rules, which are higher-level rules about how to apply lower-level rules. Examples include a meta-rule that changes the precedence of lower-level rules, or one that selects for consideration a subset of the rules, and eliminates the others as irrelevant to the current situation.

Clearly, the implementation of a rule-based tactics class can become arbitrarily large and complex, depending on how sophisticated the tactics modeling needs to be. A good way

to start would be to build something fairly simple like event-triggered rules, or perhaps a small list of precedence-ordered rules. Then the challenge will be to define the rules and the rule-interpreter in a general way so as to allow for increasingly sophisticated ways of specifying tactics.

### 6.3 Tactical Picture

Commanders need a repository in which all relevant information is integrated into a coherent “tactical picture.” This repository represents their *perception* of what is happening; it may contain incomplete, incorrect, or contradictory information. Information may be incorrect because of errors or due to latency. The `S_Commander` class has a generic `SituationObject` member for this purpose, but the class is essentially stubbed out (it just contains a list of contact reports).

What is required is a class that sorts through contact reports sent to the commander and attempts to identify what type of asset (or other entity) generated the detection, what its alliance is, and what its current state is (including its current position and where it is headed). This process, called data fusion, is a very difficult problem, and an entire area of study in its own right.

### 6.4 Plans

Any commander object can have a plan. This consists of a time-stamped list of actions that the commander intends to execute. These actions can be removed from the list and executed individually or all at once. One upgrade would be to allow executing all actions whose time-stamp was less than a given time.

Another issue concerns plan revision. As the simulation unfolds, a commander may receive information that makes him want to alter his plan. It should be convenient for the commander to add, remove, or replace actions in his plan (currently, he can only add and execute actions). Part of revising a plan entails canceling future events generated by executing actions whose time-stamp is ahead of the current time. Facilities should be built to make it easy to do this, for example by providing a method that would cancel all action events scheduled after a specific time.

The definition of a plan could be made more flexible. One upgrade would be to allow sub-plans within a plan. The definition of a plan would be a recursive one: i.e. a plan consists of time-stamped actions and/or plans. Another enhancement would be to have contingent actions (or sub-plans) in a plan. These would be executed only if (or when) a previous action succeeds. For example, a commander may wish to send in bombers to attack a target only after the target's defensive weapons and sensors have been destroyed.

Finally, it may be useful for a commander to send a plan to another commander (probably an order to a subordinate). This plan would be combined with (or replace) the recipient's current plan, and would eventually be executed.

## **6.5 Logistics**

In a war, supplying forces with food, fuel, weapons, equipment, medical supplies, parts, etc. is a huge, complex and daunting task. And yet, being able to do so effectively can be a decisive factor.

A significant enhancement to SFSS would be the inclusion of base classes for modeling logistics. In order to perform certain activities, there is a prerequisite that necessary supplies be on hand. For example, you cannot send jets out on a strike mission unless sufficient fuel and weapons are available.

This dovetails with the idea of contingent actions in a plan: one cannot perform an action unless (or until) certain other logistical actions have been accomplished (such as the delivery of fuel and weapons). In general, there are certain activities that consume supplies, and others that replenish them. Also, there are rates of consumption/replenishment associated with these activities, and possibly “granularity” levels (e.g. certain items might be packaged in fixed-sized quantities for resupply).

Another way in which logistics fits into the existing framework has to do with tactics. Logistical tactics could be formulated to anticipate what supplies are needed where, and when. Simulating the logistical aspects of warfare can be an important part of the planning process.

## **6.6 MOE Classes**

When using simulations for analysis, the “bottom line” is the collection of MOEs output by the program. These measures summarize the outcome of the scenario. An enhancement to SFSS would be to provide base classes to aid in building and collecting MOEs.